Secure Compilation

Amal Ahmed

Northeastern University

Secure Compilation

building compilers that ensure
security properties of source programs
are **preserved** in target programs

Java source code

```java
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

Java source code

```java
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

Java source code

```java
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

compiled to C target code

```c
typedef struct account_t {
  int balance = 0;
  void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount ) {
  a→balance += amount;
  return;
}
```

Java source code

```java
package Bank;

public class Account{
 private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

compiled to C target code

```c
typedef struct account_t {
 int balance = 0;
 void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount ) {
 a->balance += amount;
 return;
}
```

Java source code

```java
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

compiled to C target code

```c
typedef struct account_t {
  int balance = 0;
  void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount ) {
  a->balance += amount;
  return;
}
```

Java source code

```java
package Bank;

public class Account{
 private int balance = 0;

 public void deposit( int amount ) {
  this.balance += amount;
 }
}
```

compiled to C target code

```c
typedef struct account_t {
 int balance = 0;
 void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount ) {
 a->balance += amount;
 return;
}
```

Java source code

```
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

compiled to C target code

```
typedef struct account_t {
  int balance = 0;
  void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount ) {
  a->balance += amount;
  return;
}
```

context

Java source code

```
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

- gap between source and target abstractions

compiled to C target code

```
typedef struct account_t {
  int balance = 0;
  void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount ) {
  a->balance += amount;
  return;
}
```

context

Java source code

```
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

compiled to C target code

```
typedef struct account_t {
  int balance = 0;
  void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount ) {
  a→balance += amount;
  return;
}
```

context

- gap between source and target abstractions

- need some mechanism to hide balance in target

Java source code

```
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

compiled to C target code

```
typedef struct account_t {
  int balance = 0;
  void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount ) {
  a→balance += amount;
  return;
}
```

context

- gap between source and target abstractions

- need some mechanism to hide balance in target

- how do we prove that compiler preserves security properties ... and how are source properties expressed

More source-level abstractions and target-level attacks

Security properties as program equivalences

Security Properties as Program Equivalences

Example: Confidentiality

```
private secret : Int = 0;

public setSecret() : Int {
  secret = 1;
  return 0;
}
```

Security Properties as Program Equivalences

Example: Confidentiality

```
private secret : Int = 0;

public setSecret() : Int {
  secret = 1;
  return 0;
}
```

```
private secret : Int = 0;

public setSecret( ) : Int {
  secret = 0;
  return 0;
}
```

Security Properties as Program Equivalences

Example: Integrity

```
public proxy( callback : Unit → Unit )
      : Int {
  var secret = 0;
  callback();

  return 0;


}
```

Security Properties as Program Equivalences

Example: Integrity

```
public proxy( callback : Unit → Unit )
      : Int {
  var secret = 0;
  callback();

  return 0;


}
```

```
public proxy( callback : Unit → Unit )
      : Int {
  var secret = 0;
  callback();
  if ( secret == 0 ) {
    return 0;
  }
  return 1;
}
```

Security Properties as Program Equivalences

Example: Unbounded vs. finite memory

```
public kernel( n : Int, callback : Unit
      → Unit ) : Int {
  for (Int i = 0; i < n; i++){
   new Object();
  }
  callback();
  // security-relevant code
  return 0;
}
```

Security Properties as Program Equivalences

Example: Unbounded vs. finite memory

```
public kernel( n : Int, callback : Unit
      → Unit ) : Int {
  for (Int i = 0; i < n; i++){
   new Object();
  }
  callback();
  // security-relevant code
  return 0;
}
```

```
public kernel( n : Int, callback : Unit
      → Unit ) : Int {
  callback();
  // security-relevant code
  return 0;
}
```

Security Properties as Program Equivalences

Example: Memory Allocation Order

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
  return x;
}
```

Security Properties as Program Equivalences

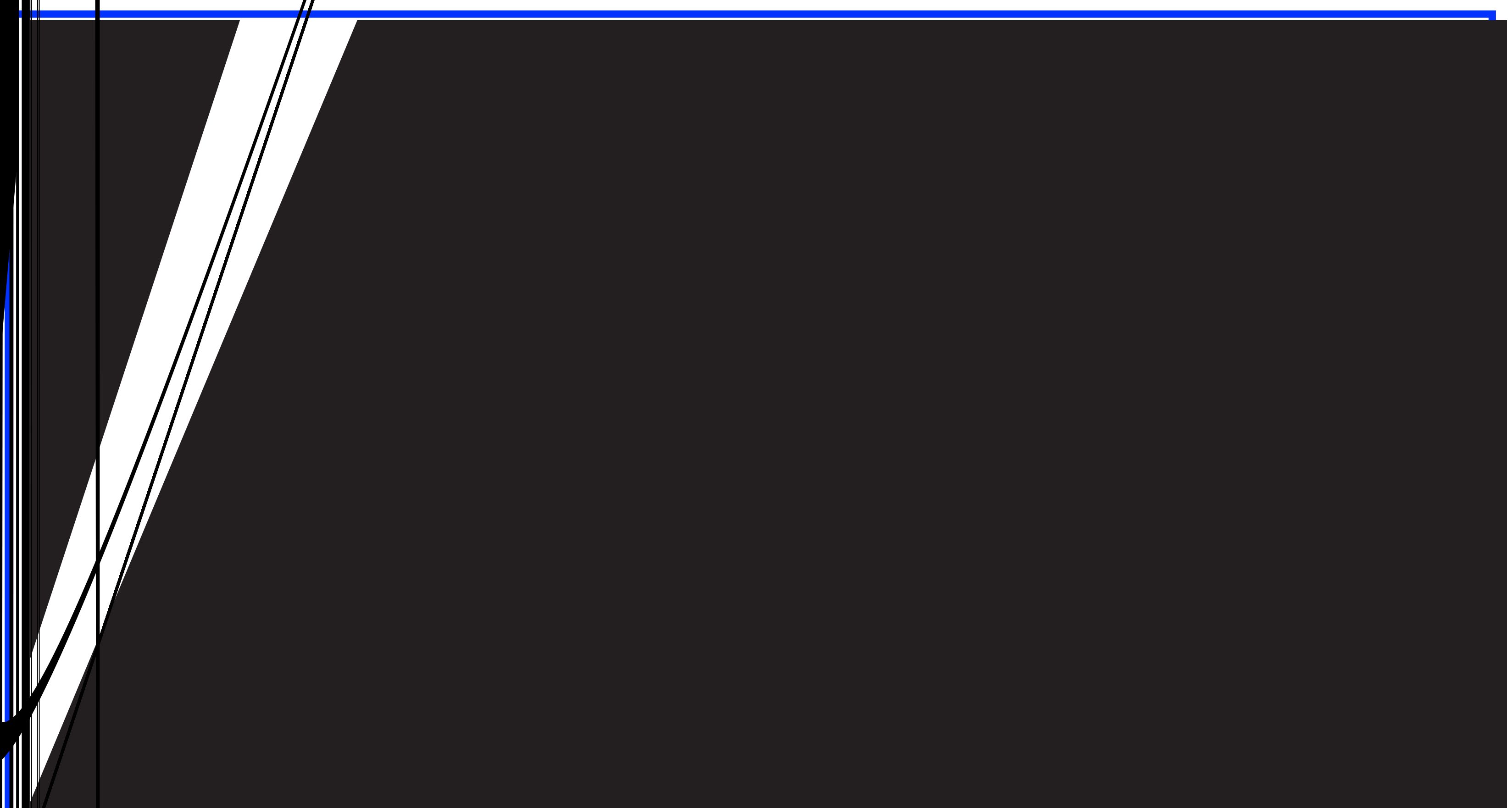
Example: Memory Allocation Order

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
  return x;
}
```

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
  return y;
}
```

Other Examples

Example: Typed source to untyped target



the code below. In this example the code provides an implementation of

Other Examples

Example: target can disrupt well-bracketed control flow when calls and returns are just jumps as in assembly

Example: target with first-class control can manipulate continuations to leak data

1. Preserving security properties expressed as some form of equivalence

- contextual equivalence

  (different for C, ML, Gallina, DSLs)

1. Preserving security properties expressed as some form of equivalence

- contextual equivalence
  (different for C, ML, Gallina, DSLs)
- observer-sensitive equivalence
  (e.g., noninterference in security-typed languages)

1. Preserving security properties expressed as some form of equivalence

- contextual equivalence
  (different for C, ML, Gallina, DSLs)
- observer-sensitive equivalence
  (e.g., noninterference in security-typed languages)
- timing/resource-sensitive equivalence
  (e.g., security of constant-time code)

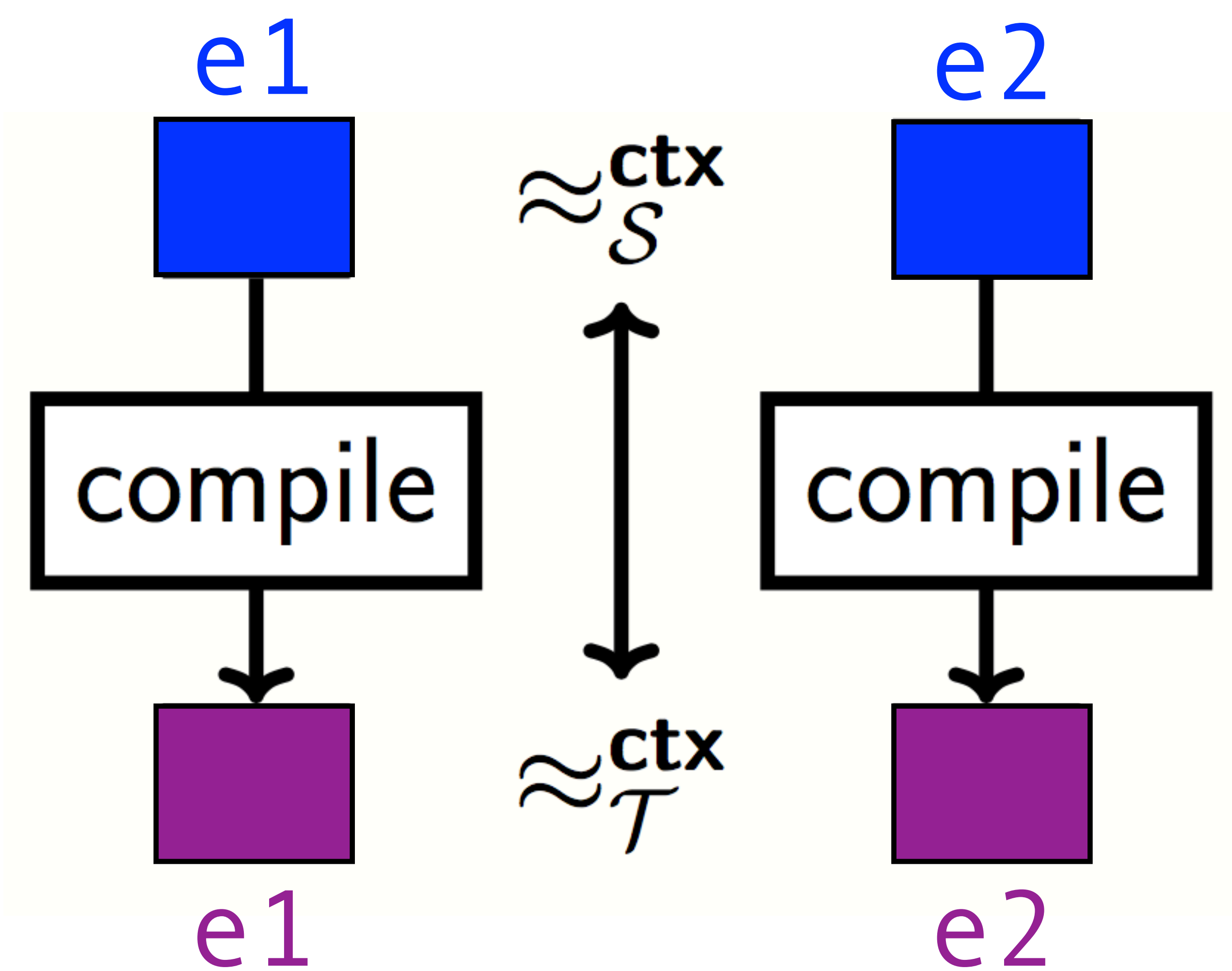
1. Preserving security by preserving equivalence
2. Different compilation targets and threat models
   - is the target language typed or untyped?
   - what observations can the attacker make?

1. Preserving security by preserving equivalence
2. Different compilation targets and threat models
   - is the target language typed or untyped?
   - what observations can the attacker make?
3. Different ways of enforcing secure compilation
   - static checking
   - dynamic checking (e.g., runtime monitoring, cryptographic & hardware enforcement)

1. Preserving security by preserving equivalence
2. Different compilation targets and threat models
   - is the target language typed or untyped?
   - what observations can the attacker make?
3. Different ways of enforcing secure compilation
   - static checking
   - dynamic checking (e.g., runtime monitoring, cryptographic & hardware enforcement)
4. Proof techniques
   - "back-translating" target attackers to source

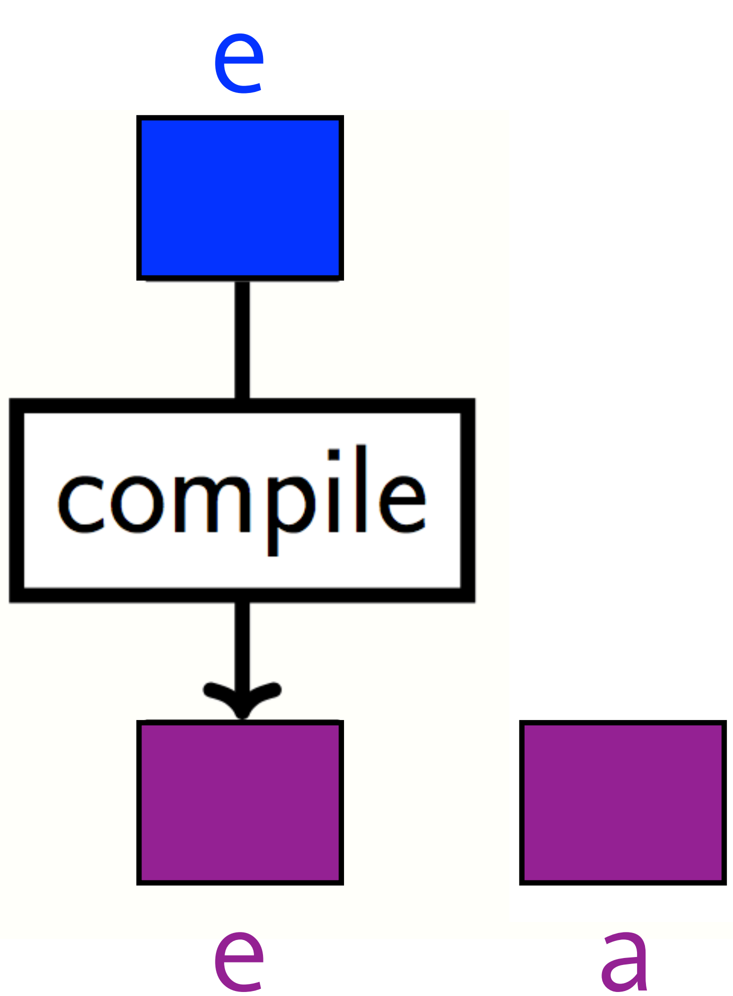
Fully Abstract Compilation

Preserve and reflect contextual equivalence



Fully Abstract Compilation

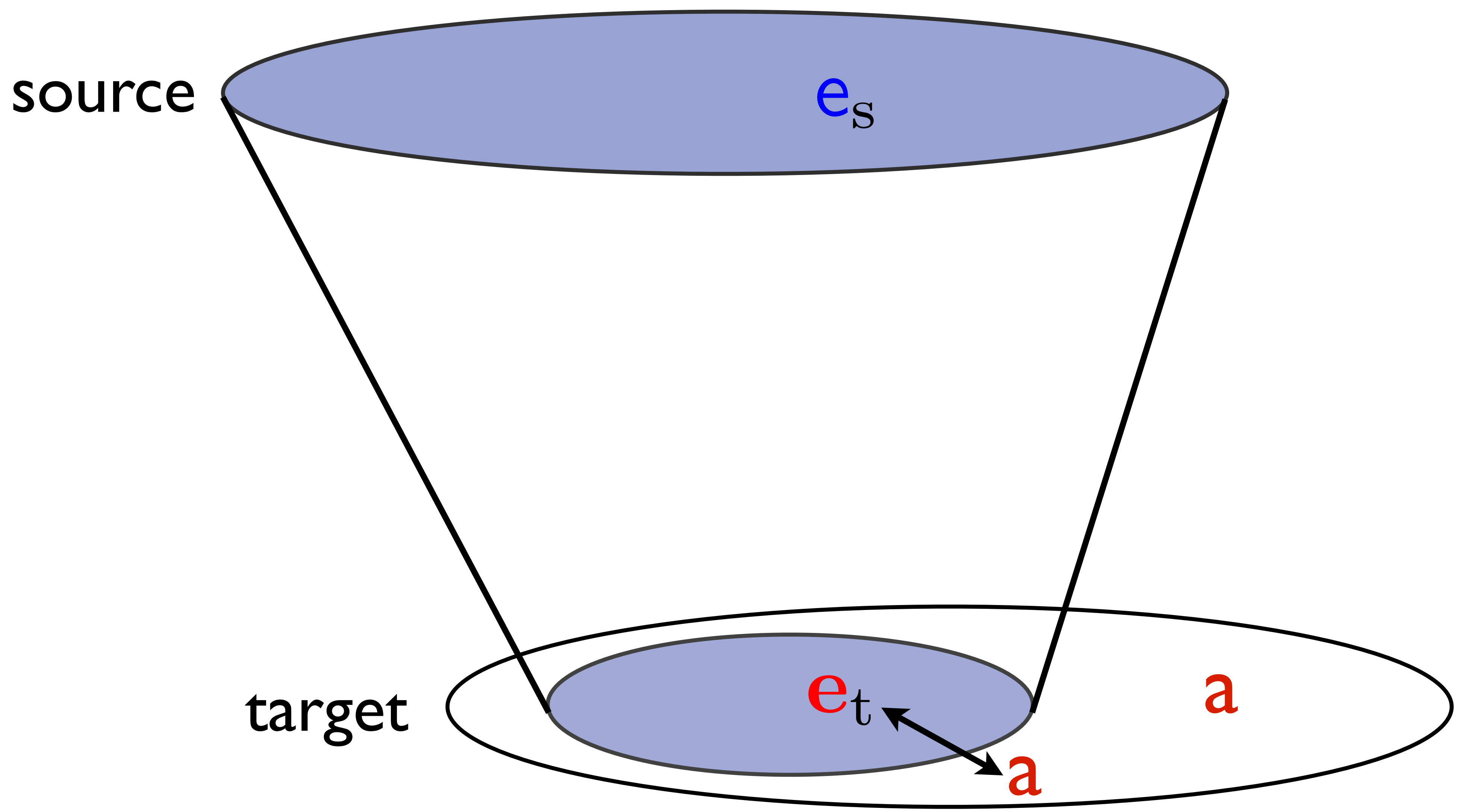
Preserve contextual equivalence



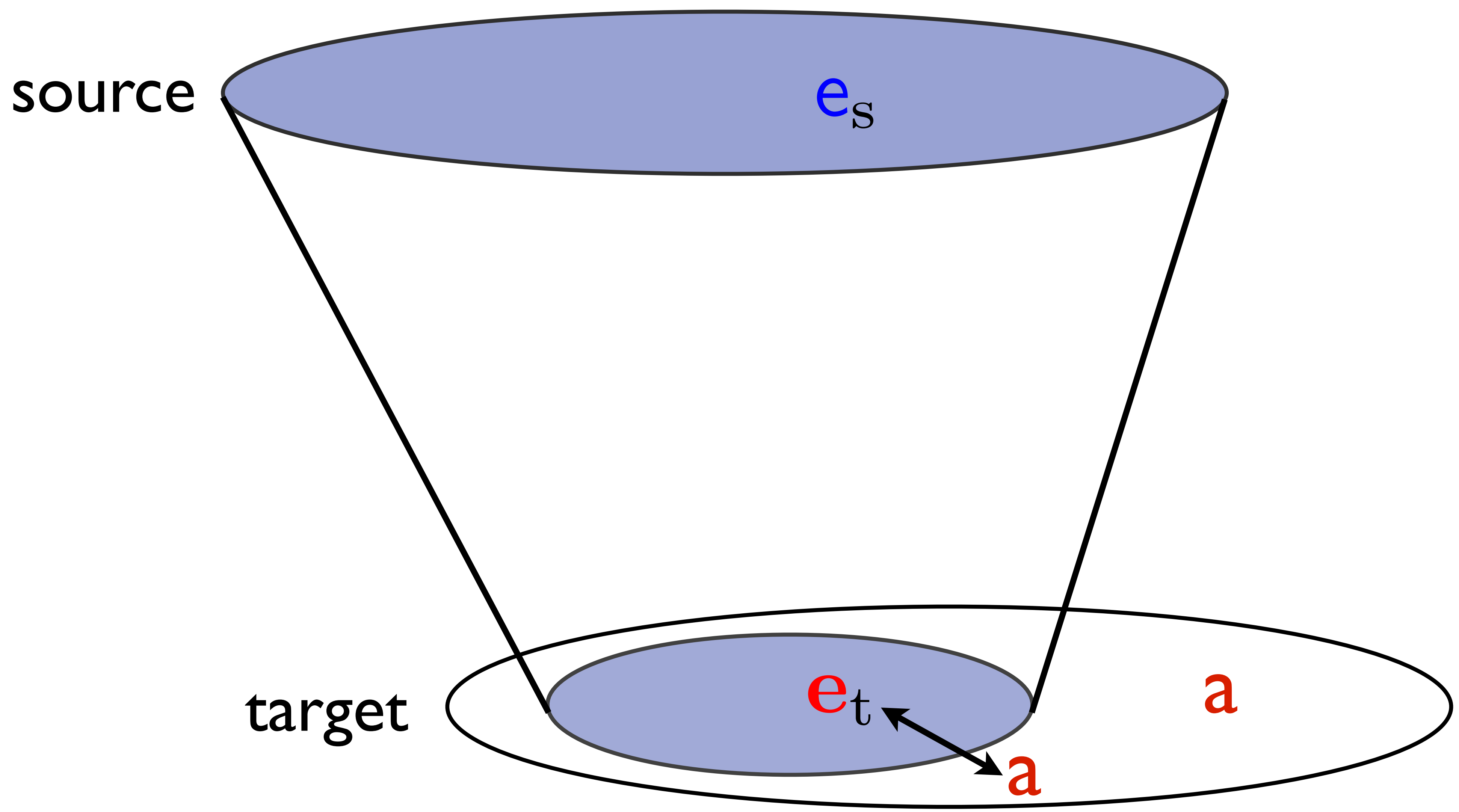
Guarantees that e will remain as secure as e when executed in arbitrary target-level contexts

i.e. target contexts (attackers a) can make no more observations about e than a source context can make about e

Ensuring Full Abstr. / Secure Comp.

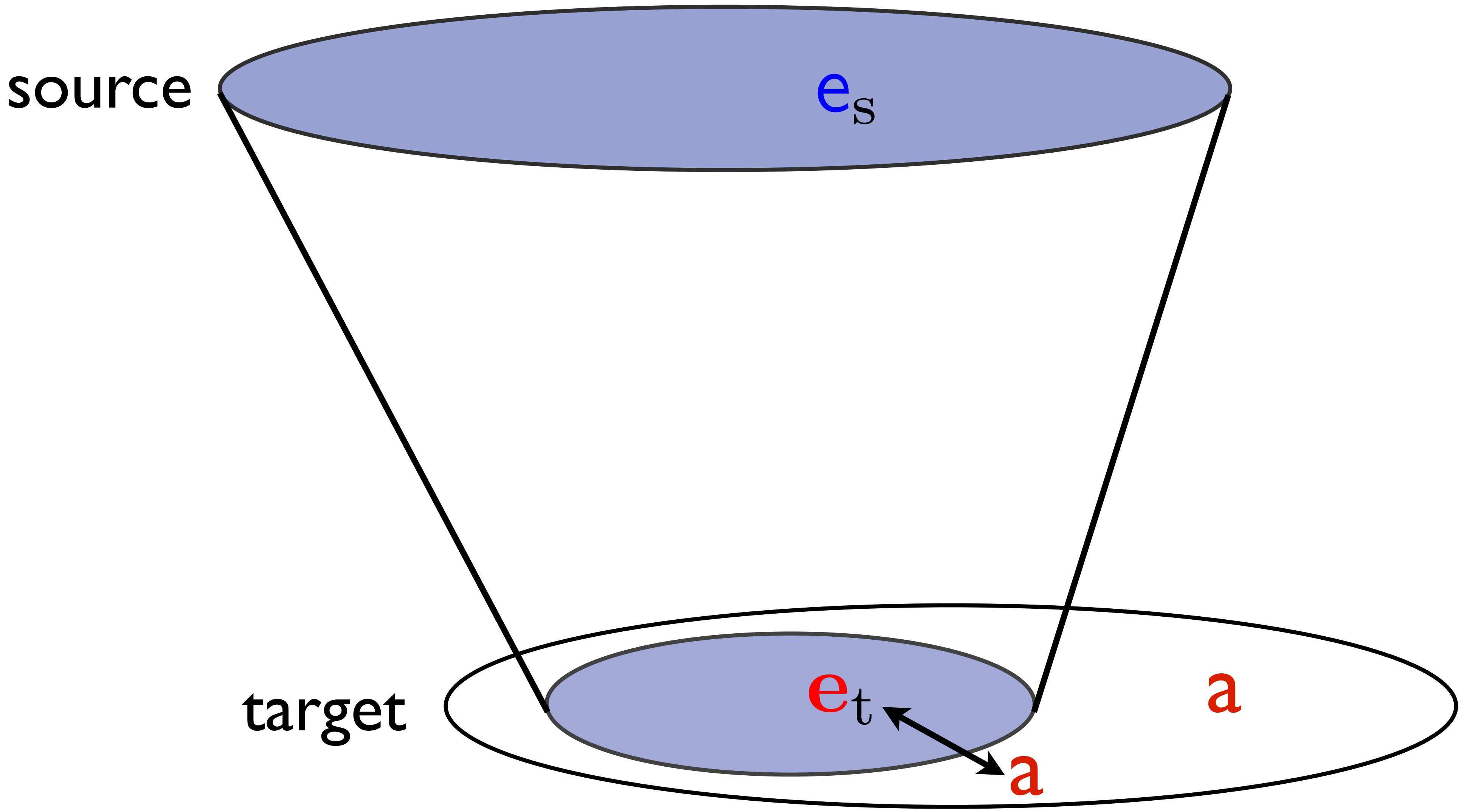


Ensuring Full Abstr. / Secure Comp.

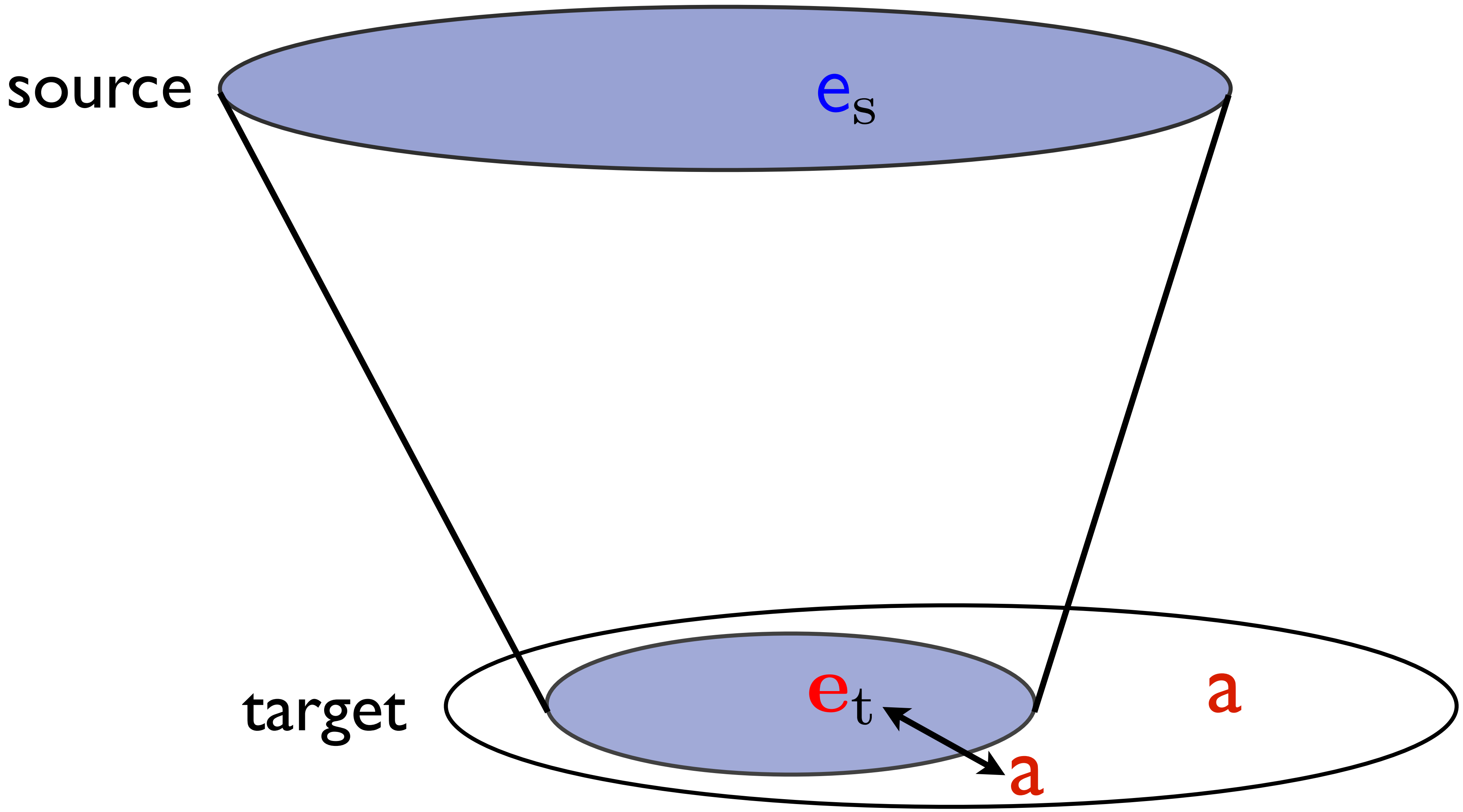


Must ensure that any a we link with behaves like some source context

Ensuring Full Abstr. / Secure Comp.

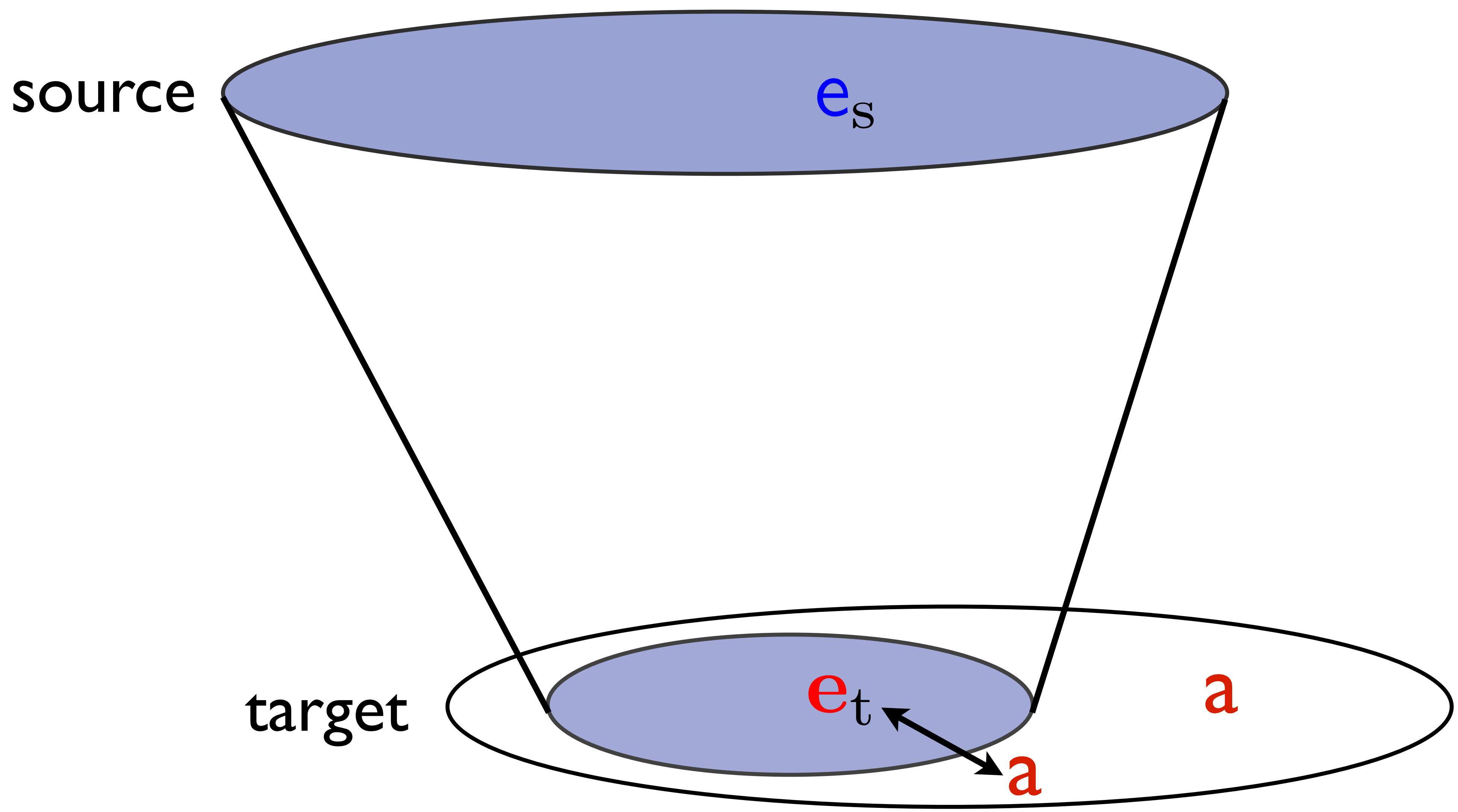


Ensuring Full Abstr. / Secure Comp.



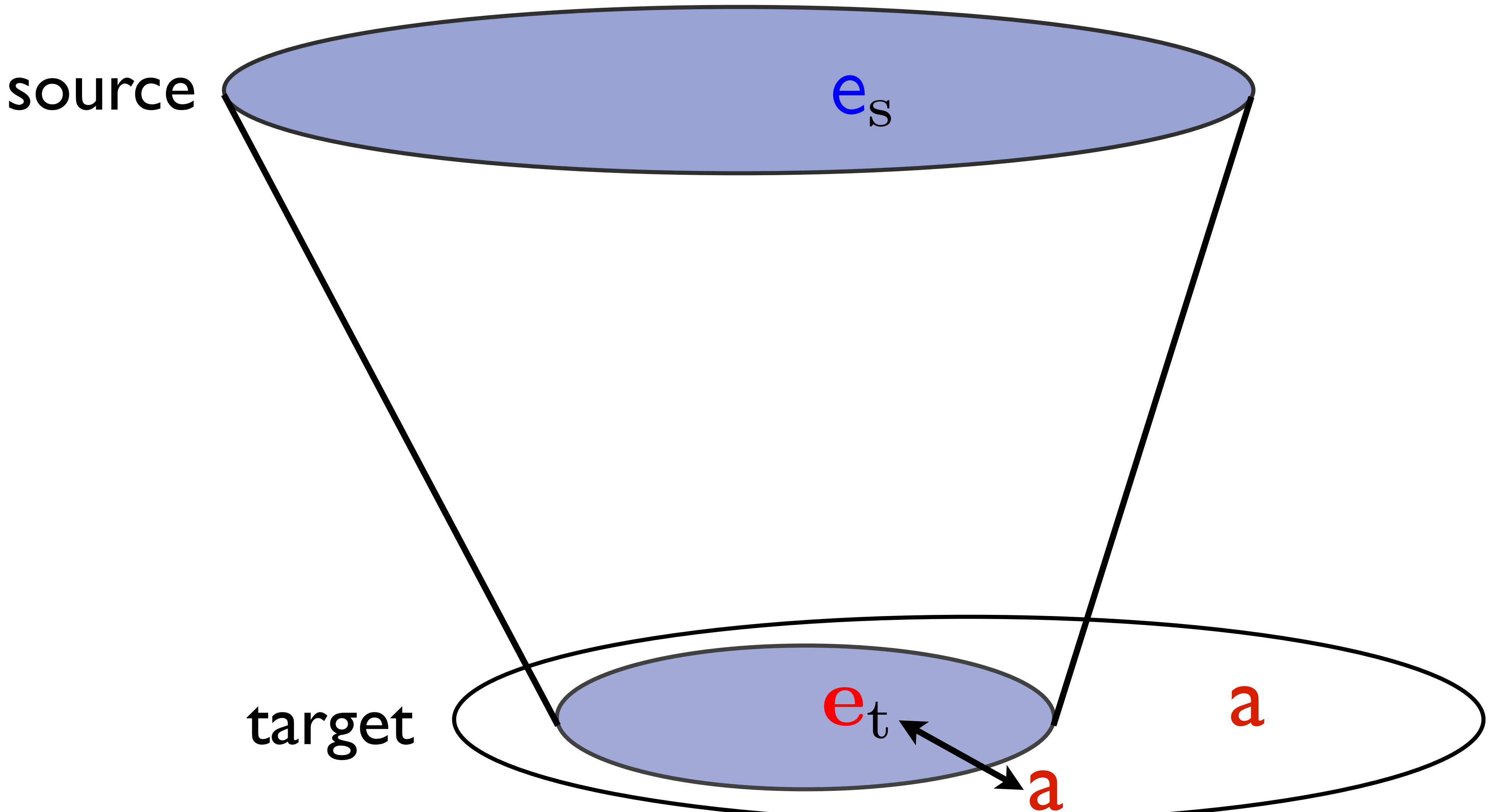
1. Add target features to the source language.

Ensuring Full Abstr. / Secure Comp.



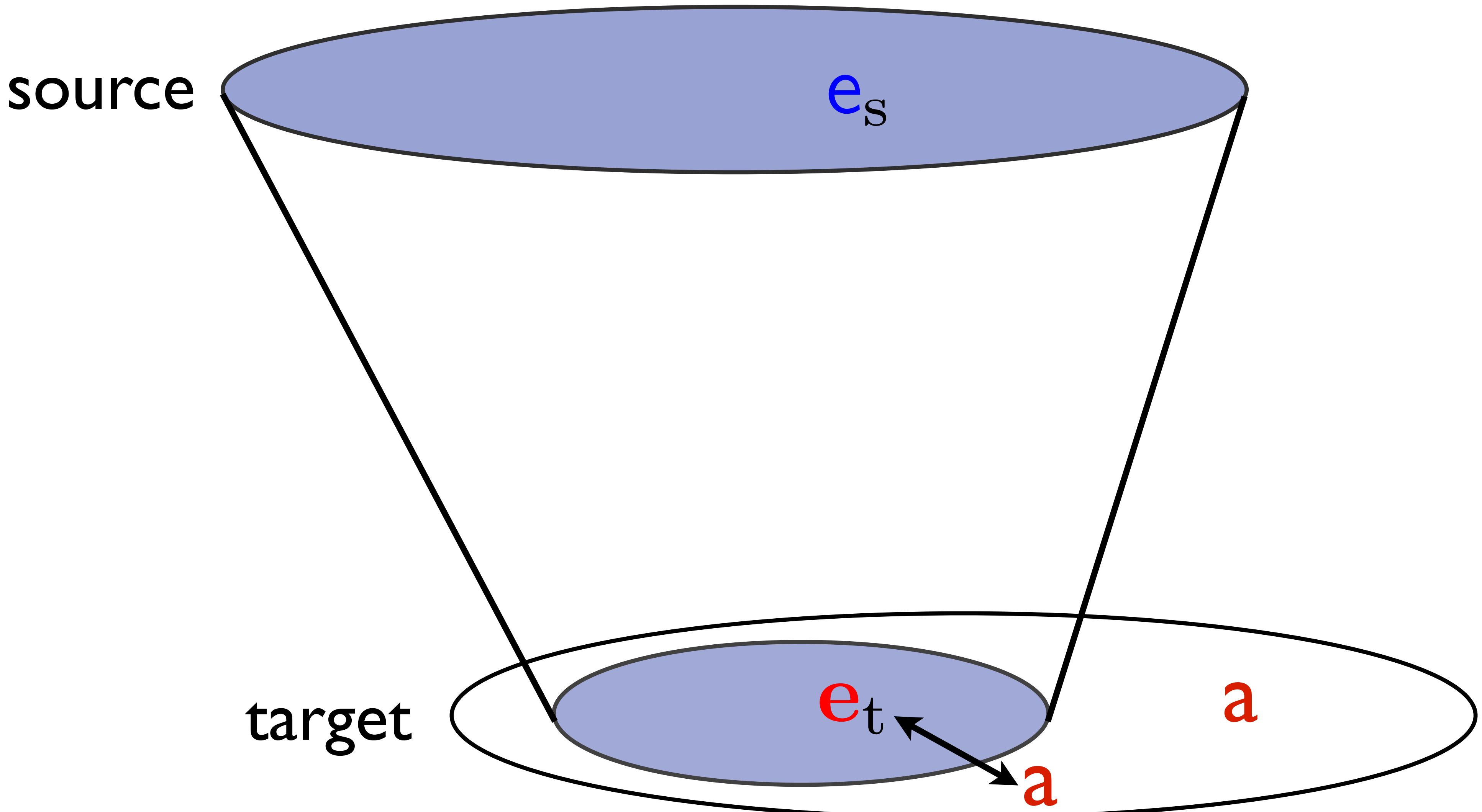
1. Add target features to the source language. **Bad!**

Ensuring Full Abstr. / Secure Comp.



1. Add target features to the source language. **Bad!**
2. Dynamics checks: catch badly behaved code in the act.

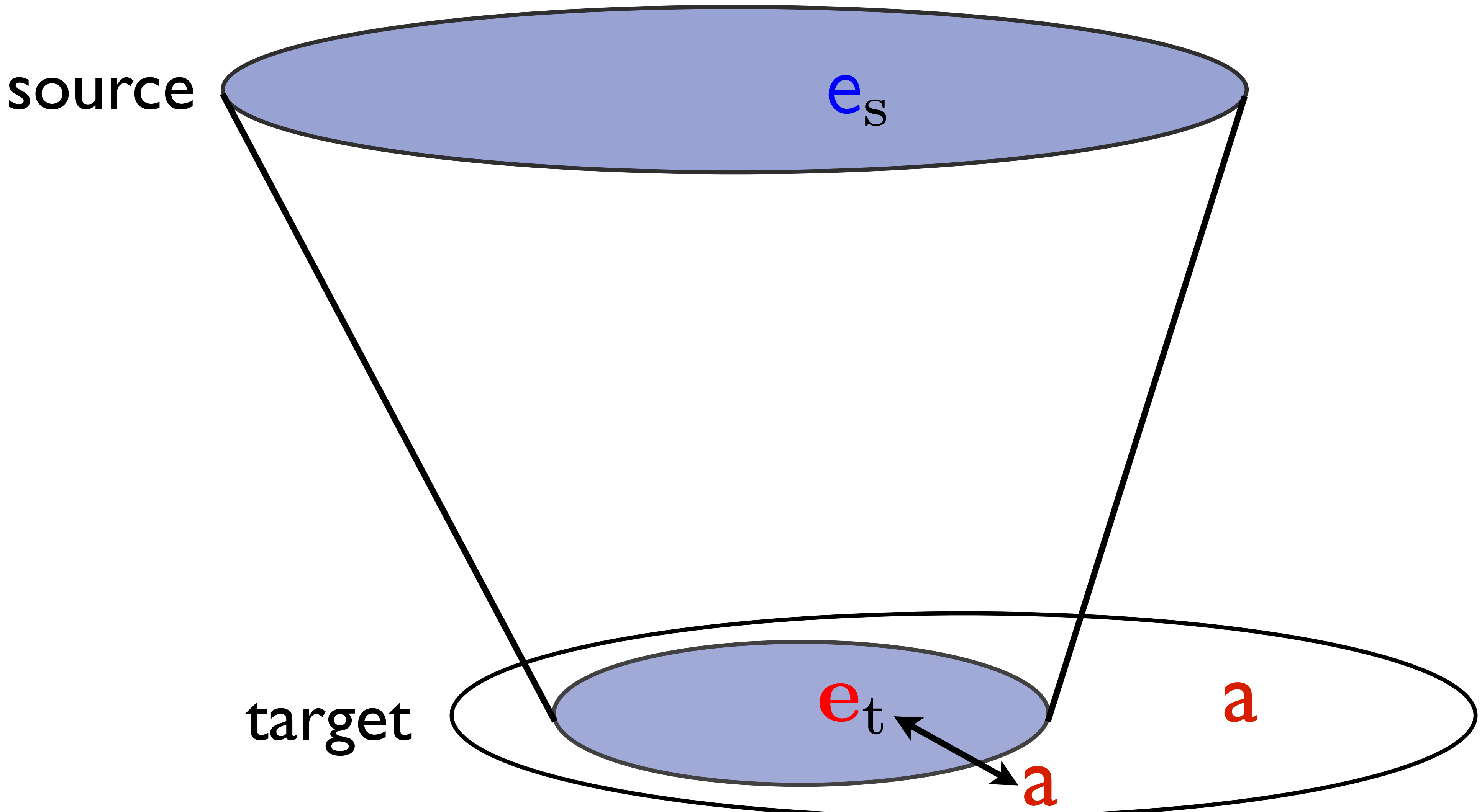
Ensuring Full Abstr. / Secure Comp.



1. Add target features to the source language. **Bad!**
2. Dynamics checks: catch badly behaved code in the act.

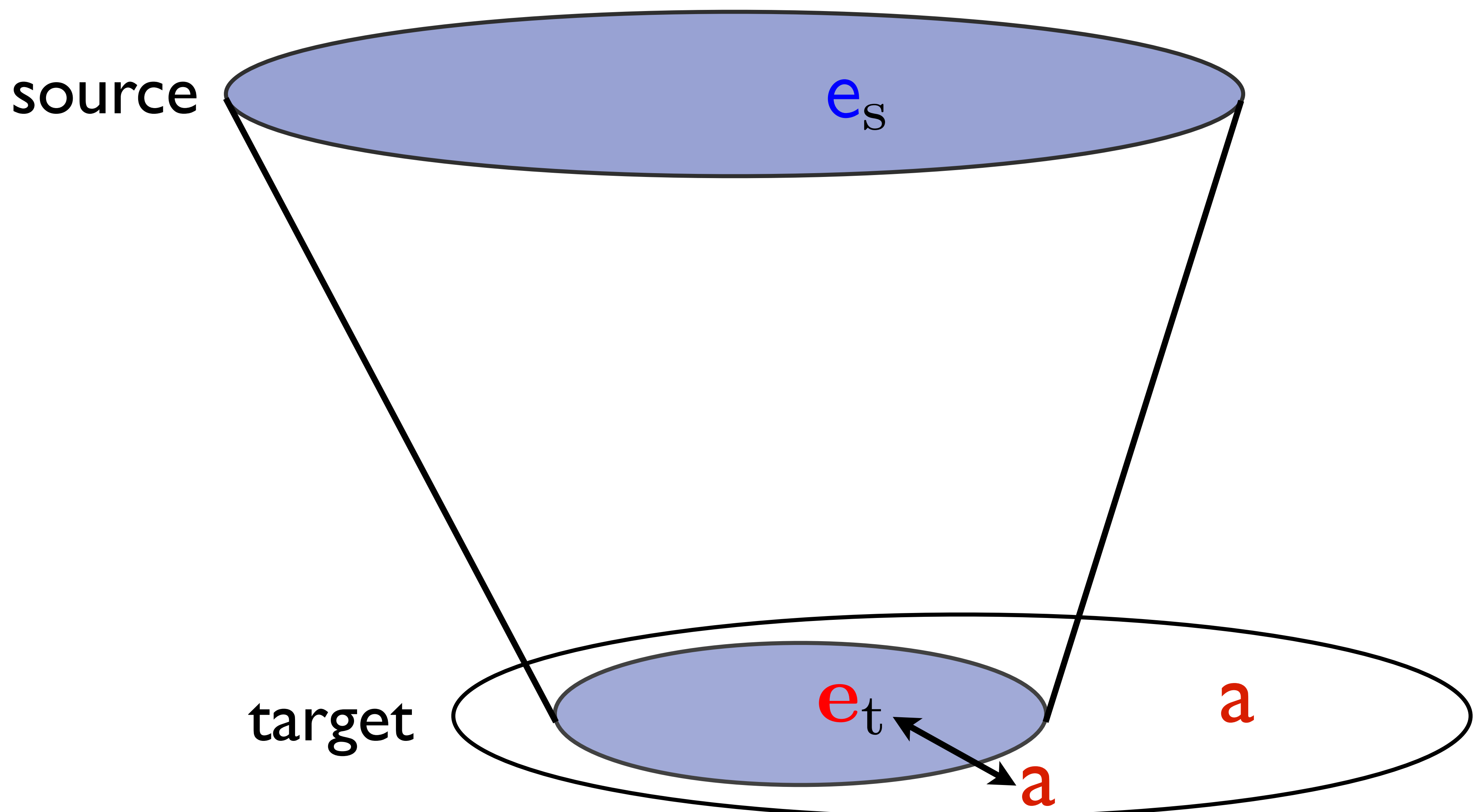
Performance cost

Ensuring Full Abstr. / Secure Comp.



1. Add target features to the source language. **Bad!**
2. Dynamics checks: catch badly behaved code in the act.
   Performance cost
3. Static checks: rule our badly behaved code in the first place

Ensuring Full Abstr. / Secure Comp.



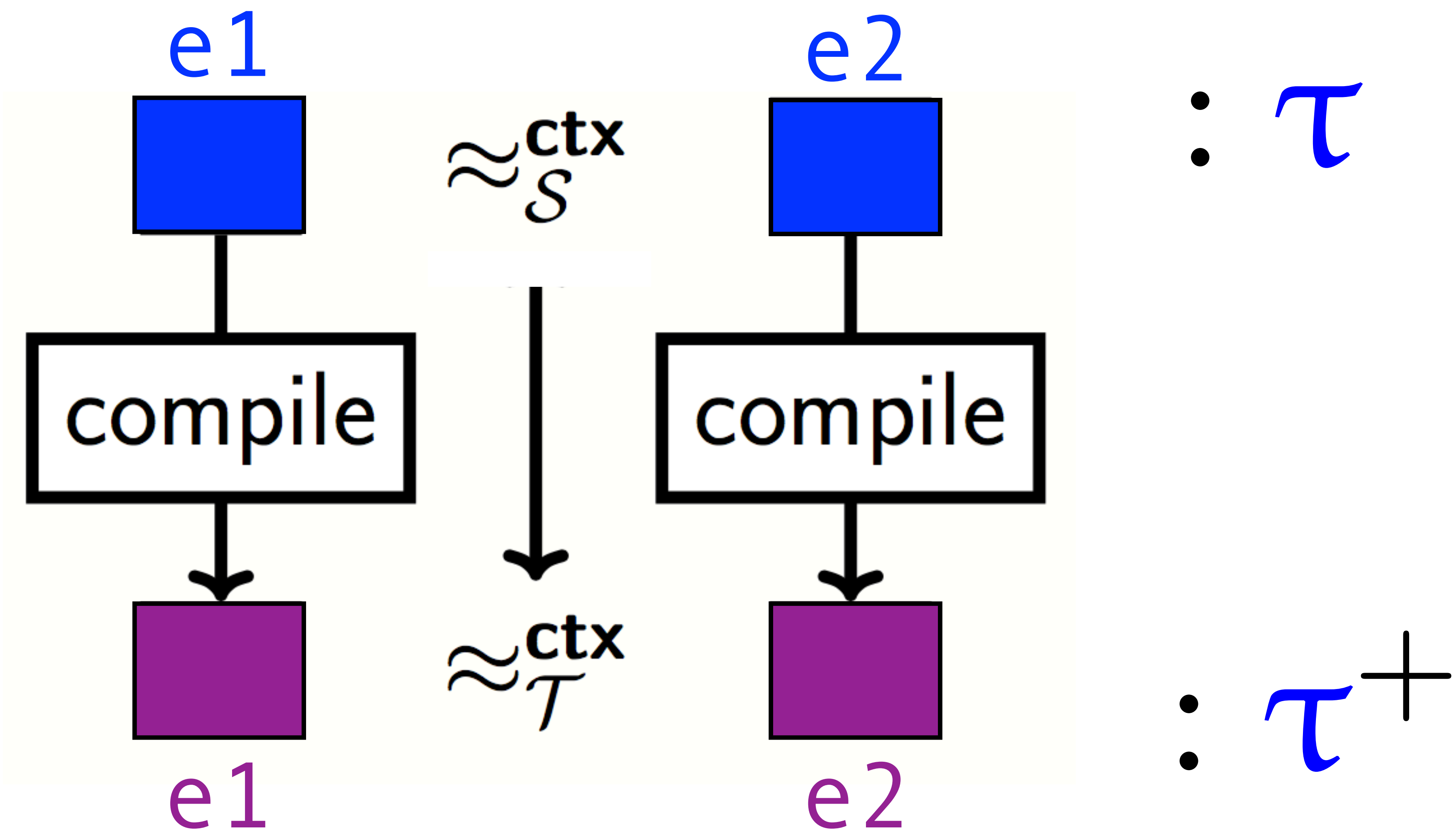
1. Add target features to the source language. **Bad!**
2. Dynamics checks: catch badly behaved code in the act. **Performance cost**
3. Static checks: rule our badly behaved code in the first place **Verification**

Type-Preserving Compilation

$$e : \tau \;\leadsto\; e : \tau^{+}$$

Type-Preserving Secure Compilation

Preserve well-typedness & equivalence



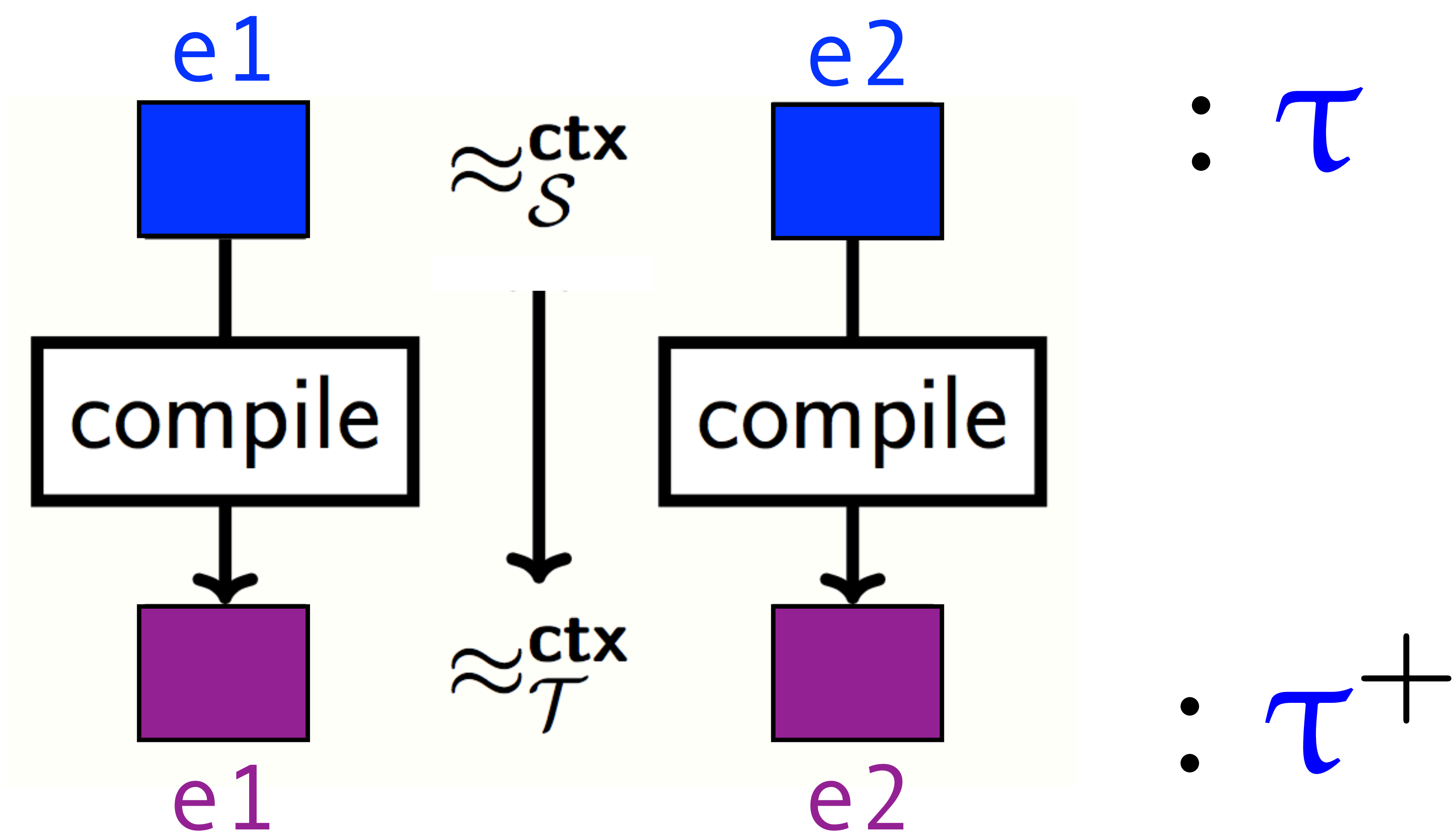
Type-Preserving Compilation

$$e : \tau \leadsto e : \tau^{+}$$

- System F to Typed Assembly Language
  [Morrisett et al. POPL'97, TOPLAS'98]
- Typed compilation of Featherweight Java to F-omega, private fields to existential type *[League et al. TOPLAS'02]*
- FINE (F# with refinement & affine types) to DCIL (dependent CIL) *[Chen et al. PLDI'10]*
- Security-type-preserving compilation from WHILE lang. to stack-based TAL (both languages satisfy noninterference). Extended to concurrent setting with thread creation, secure scheduler *[Barthe et al. 2007, 2010]*

Type-Preserving Secure Compilation

Preserve well-typedness & equivalence



Challenge: Proving Full Abstraction

Suppose $\Gamma \vdash \mathsf{e}_1 : \tau \rightsquigarrow \mathsf{e}_1$ and $\Gamma \vdash \mathsf{e}_2 : \tau \rightsquigarrow \mathsf{e}_2$

$$\Gamma \vdash \mathsf{e}_1 \approx_{\mathrm{S}}^{ctx} \mathsf{e}_2 : \tau$$

$$\Downarrow$$

$$\Gamma^+ \vdash \mathsf{e}_1 \approx_{\mathrm{T}}^{ctx} \mathsf{e}_2 : \tau^+$$

Challenge: Proving Full Abstraction

Suppose $\Gamma \vdash \mathsf{e}_1 : \tau \rightsquigarrow \mathsf{e}_1$ and $\Gamma \vdash \mathsf{e}_2 : \tau \rightsquigarrow \mathsf{e}_2$

$$\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \tau$$

Given:
No C_{S} can distinguish e_1, e_2

$$\downarrow$$

$$\Gamma^{+} \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{T}} \mathsf{e}_2 : \tau^{+}$$

Challenge: Proving Full Abstraction

Suppose $\Gamma \vdash \mathsf{e}_1 : \tau \rightsquigarrow \mathsf{e}_1$ and $\Gamma \vdash \mathsf{e}_2 : \tau \rightsquigarrow \mathsf{e}_2$

$$\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \tau$$

$$\downarrow$$

$$\Gamma^+ \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{T}} \mathsf{e}_2 : \tau^+$$

Given:
No C_{S} can distinguish e_1, e_2

Show:
Given arbitrary C_{T} it cannot distinguish e_1, e_2

Challenge: Proving Full Abstraction

Suppose $\Gamma \vdash \mathsf{e}_1 : \tau \rightsquigarrow \mathsf{e}_1$ and $\Gamma \vdash \mathsf{e}_2 : \tau \rightsquigarrow \mathsf{e}_2$

$$\Gamma \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{S}} \mathsf{e}_2 : \tau$$

$$\Big\downarrow$$

$$\Gamma^+ \vdash \mathsf{e}_1 \approx^{ctx}_{\mathrm{T}} \mathsf{e}_2 : \tau^+$$

Given:

No C_{S} can distinguish e_1, e_2

Show:

Given arbitrary C_{T} it cannot distinguish e_1, e_2

Need to be able to "back-translate" C_{T} to an equivalent C_{S}

Closure Conversion

On board: Translation Correctness and Full Abstraction